Assignment-1: Shortest Path Finding on a Map

Md. Asad Mondall [20CSE006]

Introduction: The project is to determine the shortest way between two points (Barishal-Current Location and Home) on this given map. The map composes a number of nodes for positions and edges which are roads between them. The project is based on two algorithms, namely Depth-First Search (DFS) and Breadth-First Search (BFS), which discover the shortest route from the starting point to the destination point.

Algorithm Functionalities:

Depth-First Search (DFS):

- DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.
- In the context of this project, DFS is used to find the shortest path from the starting location to the destination recursively.
- It explores each neighboring location recursively until it reaches the destination, maintaining the path taken so far.
- If the destination is reached, the algorithm returns the path. Otherwise, it backtracks and explores other paths.

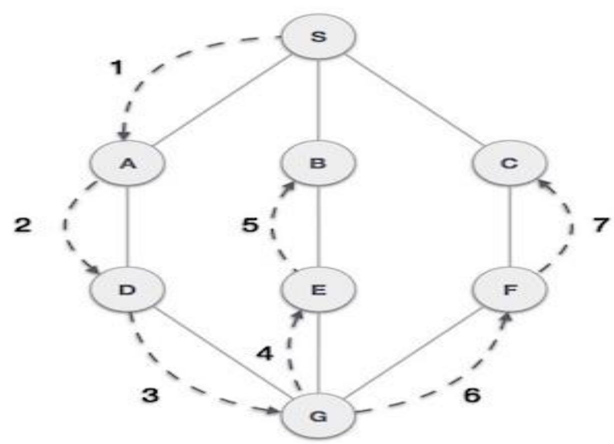

Figure: DFS Functionality

Source:[*https://www.tutorialspoint.com/data_structures_algorithms/images/depth_first_traversal.jpg*](https://www.tutorialspoint.com/data_structures_algorithms/images/depth_first_traversal.jpg)

Breadth-First Search (BFS):

- BFS is a graph traversal algorithm that explores all neighbor nodes at the present depth prior to moving on to nodes at the next depth level.
- In this project, BFS is used to find the shortest path from the starting location to the destination iteratively.

- It starts by exploring all immediate neighbors of the starting location, then explores their neighbors, and so on, until it reaches the destination.
- BFS guarantees the shortest path in terms of the number of edges traversed.

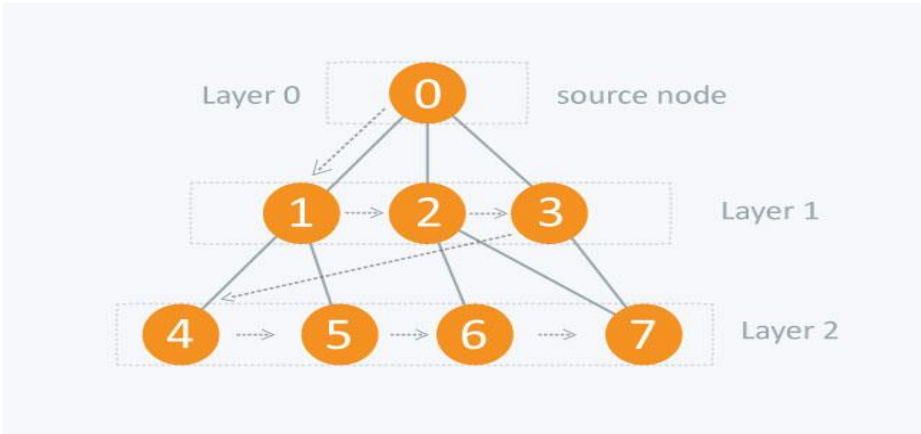

Figure: BFS Functionality

Source: https://he-s3.s3.amazonaws.com/media/uploads/fdec3c2.jpg

Map Generation:

- The map is represented as a graph, where nodes represent locations and edges represent the roads connecting them.
- Each node has coordinates associated with it for visualization purposes.
- The edges between nodes are weighted with the distance between the locations they connect.

#map.py

```
import networkx as nx

import matplotlib.pyplot as plt

nodes = ["Barishal", "Tekerhat", "Faridpur", "Magura", "Arpara", "Home", "Khulna", "Shalikha", "Jashore", "Gopalganj", "Kalna", "Narail"]

edges = [("Barishal", "Tekerhat", 76.6), ("Tekerhat", "Faridpur", 51.1), ("Faridpur", "Magura", 51.1), ("Magura", "Arpara", 15),

        ("Arpara", "Home", 1.1), ("Barishal", "Khulna", 113), ("Khulna", "Shalikha", 67.7), ("Shalikha", "Arpara", 9.5),

        ("Khulna", "Jashore", 60), ("Jashore", "Arpara", 31.6), ("Tekerhat", "Gopalganj", 115.4), ("Gopalganj", "Kalna", 41),

        ("Kalna", "Narail", 27), ("Narail", "Arpara", 45.2)]
```

```
G = nx.Graph()
G.add_nodes_from(nodes)

for edge in edges:
    G.add_edge(edge[0], edge[1], weight=edge[2])

pos = {
    "Barishal": (1, 1),
    "Tekerhat": (2, 2),
    "Faridpur": (3, 2),
    "Magura": (3, 1),
    "Arpara": (2.5, 0),
    "Home": (2.5, -0.5),  # Adjusted position for "Home" node
    "Khulna": (4, 2),
    "Shalikha": (4.5, 1),
    "Jashore": (5, 1.5),
    "Gopalganj": (2, 3),
    "Kalna": (2.5, 3.5),
    "Narail": (3, 4)
}

plt.figure(facecolor='grey')  # Set background color
nx.draw_networkx_nodes(G, pos, node_size=700, node_color='green')
nx.draw_networkx_edges(G, pos, width=2)
nx.draw_networkx_labels(G, pos, font_size=10, font_color='yellow', font_family="sans-serif")
edge_labels = {(edge[0], edge[1]): str(edge[2]) for edge in edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')
plt.title("Map Visualization\n Md. Asad Mondall [20CSE006]", color='darkorange')
plt.axis("off")
plt.show()
```

Visualization:

- The map, along with the shortest path, is visualized using Matplotlib.
- Nodes representing locations are displayed as circles, with labels indicating the names of the locations.
- Edges representing roads are drawn between nodes, with labels indicating the distances.
- The shortest path found by either DFS or BFS is highlighted with a bold line on the map for better visualization.

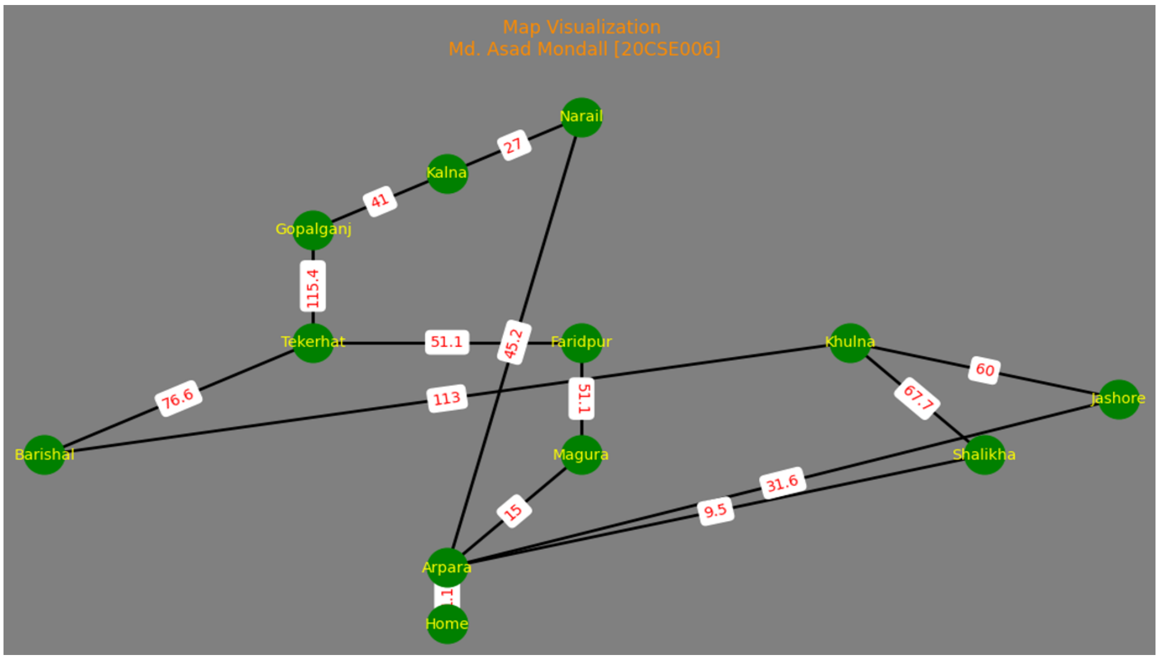

Figure: Map

Code implementation:

#shortest_path.py

```
import networkx as nx

import matplotlib.pyplot as plt

from collections import deque

map_graph = {

    "Barishal": ["Tekerhat", "Khulna"],

    "Tekerhat": ["Barishal", "Faridpur", "Gopalganj"],

    "Faridpur": ["Tekerhat", "Magura"],

    "Magura": ["Faridpur", "Arpara"],

    "Arpara": ["Magura", "Home", "Shalikha", "Narail"],
```

```
    "Home": ["Arpara"],
    "Khulna": ["Barishal", "Shalikha", "Jashore"],
    "Shalikha": ["Khulna", "Arpara"],
    "Jashore": ["Khulna", "Arpara"],
    "Gopalganj": ["Tekerhat", "Kalna"],
    "Kalna": ["Gopalganj", "Narail"],
    "Narail": ["Kalna", "Arpara"]
}

def dfs_shortest_path(graph, start, goal, path=None):
    if path is None:
        path = [start]
    if start == goal:
        return path
    shortest_path = None
    for neighbor in graph.get(start, []):
        if neighbor not in path:
            new_path = dfs_shortest_path(graph, neighbor, goal, path + [neighbor])
            if new_path:
                if shortest_path is None or len(new_path) < len(shortest_path):
                    shortest_path = new_path
    return shortest_path

def bfs_shortest_path(graph, start, goal):
    queue = deque([[start]])
    visited = set()

    while queue:
        path = queue.popleft()
        node = path[-1]
```

```
        if node == goal:
            return path
        if node not in visited:
            visited.add(node)
            for adjacent in graph.get(node, []):
                new_path = list(path)
                new_path.append(adjacent)
                queue.append(new_path)

start_location = "Barishal"
home_location = "Home"

dfs_path = dfs_shortest_path(map_graph, start_location, home_location)
bfs_path = bfs_shortest_path(map_graph, start_location, home_location)

print("Shortest path from", start_location, "to", home_location, "using DFS:", dfs_path)
print("Shortest path from", start_location, "to", home_location, "using BFS:", bfs_path)

nodes = ["Barishal", "Tekerhat", "Faridpur", "Magura", "Arpara", "Home", "Khulna", "Shalikha", "Jashore",
"Gopalganj", "Kalna", "Narail"]
edges = [("Barishal", "Tekerhat", 76.6), ("Tekerhat", "Faridpur", 51.1), ("Faridpur", "Magura", 51.1), ("Magura",
"Arpara", 15),
         ("Arpara", "Home", 1.1), ("Barishal", "Khulna", 113), ("Khulna", "Shalikha", 67.7), ("Shalikha", "Arpara", 9.5),
         ("Khulna", "Jashore", 60), ("Jashore", "Arpara", 31.6), ("Tekerhat", "Gopalganj", 115.4), ("Gopalganj", "Kalna",
41),
         ("Kalna", "Narail", 27), ("Narail", "Arpara", 45.2)]
G = nx.Graph()
G.add_nodes_from(nodes)
for edge in edges:
    G.add_edge(edge[0], edge[1], weight=edge[2])
pos = {
    "Barishal": (1, 1),
```

```
    "Tekerhat": (2, 2),
    "Faridpur": (3, 2),
    "Magura": (3, 1),
    "Arpara": (2.5, 0),
    "Home": (2.5, -0.5),
    "Khulna": (4, 2),
    "Shalikha": (4.5, 1),
    "Jashore": (5, 1.5),
    "Gopalganj": (2, 3),
    "Kalna": (2.5, 3.5),
    "Narail": (3, 4)
}
plt.figure(facecolor='grey')
nx.draw_networkx_nodes(G, pos, node_size=700, node_color='green')
nx.draw_networkx_edges(G, pos, width=2)
nx.draw_networkx_labels(G, pos, font_size=10, font_color='yellow', font_family="sans-serif")
edge_labels = {(edge[0], edge[1]): str(edge[2]) for edge in edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')
shortest_path_edges = [(dfs_path[i], dfs_path[i + 1]) for i in range(len(dfs_path) - 1)]
nx.draw_networkx_edges(G, pos, edgelist=shortest_path_edges, width=4, edge_color='blue')
plt.title("Map Visualization\n Md. Asad Mondall [20CSE006]", color='darkorange')
plt.axis("off")
plt.show()
```

Result:

Applying DFS and BFS on the map, both algorithms determine the same shortest path from Barishal (current location) to Home location which is,

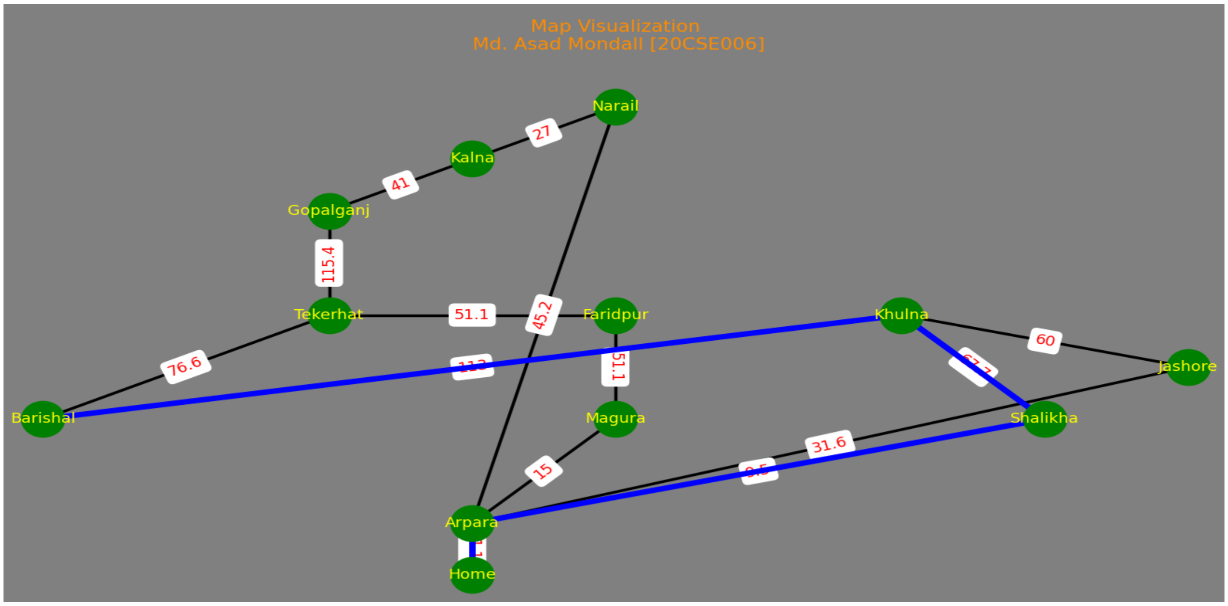

Figure: Shortest Path (blue marked path)

Shortest path from Barishal to Home using DFS: ['Barishal', 'Khulna', 'Shalikha', 'Arpara', 'Home']

Shortest path from Barishal to Home using BFS: ['Barishal', 'Khulna', 'Shalikha', 'Arpara', 'Home']

Conclusion:

The project successfully uses both DFS and BFS algorithms to find the shortest route on the given map. It offers a user-friendly graphical layout of the map and the shortest way to the destination that facilitates the understanding of the route. Further steps can be taken to enhance the algorithms in terms of handling larger maps and adding more features such as real-time traffic updates.

Assignment-2: Constraint satisfaction problems (CSPs)

1. Backtracking Search Algorithm

The backtracking search algorithm is a systematic way for finding the solution space of a specific problem and with the aim of getting a feasible solution. It is often employed in many different types of combinatorial optimization problems like the constraint satisfaction problems (CSP) and the constraint optimization problems.

How Does a Backtracking Algorithm Work?

backtracking algorithm, the algorithm seeks a path to a feasible solution that includes some intermediate checkpoints. If the checkpoints do not lead to a viable solution, the problem can return to the checkpoints and take another path to find a solution. Consider the following scenario:

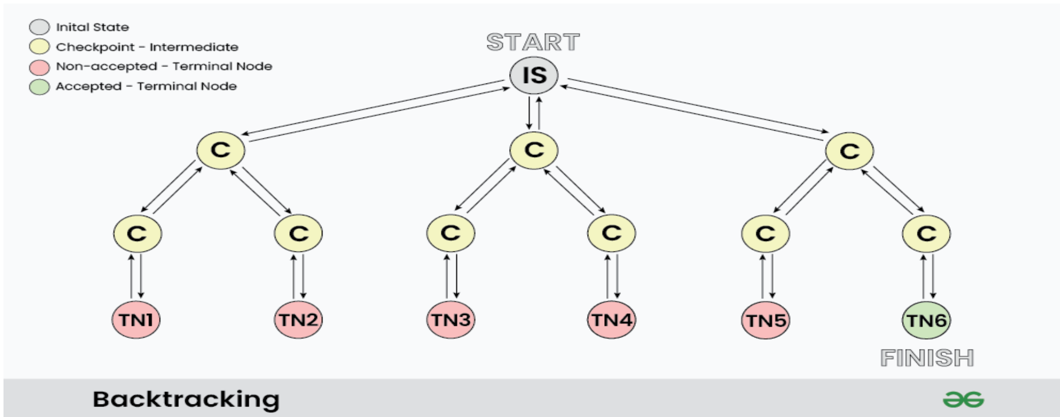

Image source: https://media.geeksforgeeks.org/wpcontent/uploads/20231010124142/backtracking.png

As shown in the figure,

IS: It represents the Initial State where the recursion call starts to find a valid solution.

C: it represents different Checkpoints for recursive calls.

TN: It represents the Terminal Nodes where no further recursive calls can be made, these nodes act as base case of recursion and we determine whether the current solution is valid or not at this state.

At each Checkpoint, the program makes some decisions and rehabilitates back from one checkpoint to other until it arrives at the terminal Node where it determines whether the solution is feasible or not after that it starts to revert back to the checkpoints and try out other paths. Take into account of such example in figure above where TN1...TN5 represents terminal nodes where the solution is not right, while TN6 is a state of problem, we have obtained there a valid solution.

The back arrows in the figure shows backtracking in actions, where we revert the changes made by some checkpoint.

Pseudocode for Backtracking:

void FIND_SOLUTIONS(parameters):

if (valid solution):

store the solution

Return

for (all choice):

if (valid choice):

APPLY (choice)

FIND_SOLUTIONS (parameters)

BACKTRACK (remove choice)

Return

Example: Map Coloring

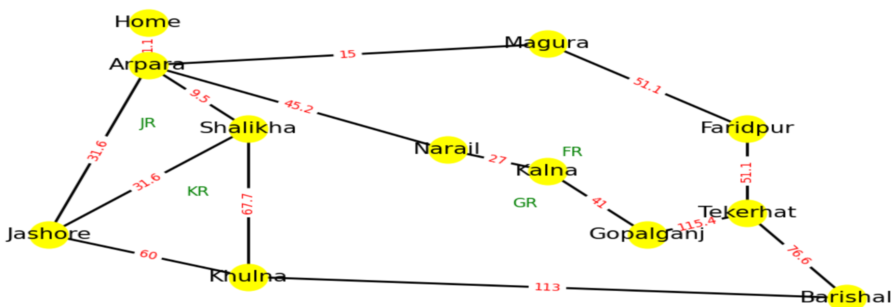

Figure: Map, before coloring

Variables: JR, KR, GR, FR

JR = Jashore Region, KR = Khulna Region, FR = Faridpur Region

Domains: {red, green, blue}

Constraints: Adjacent regions must have different colors e.g., JR ≠ KR or (JR, KR) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}.

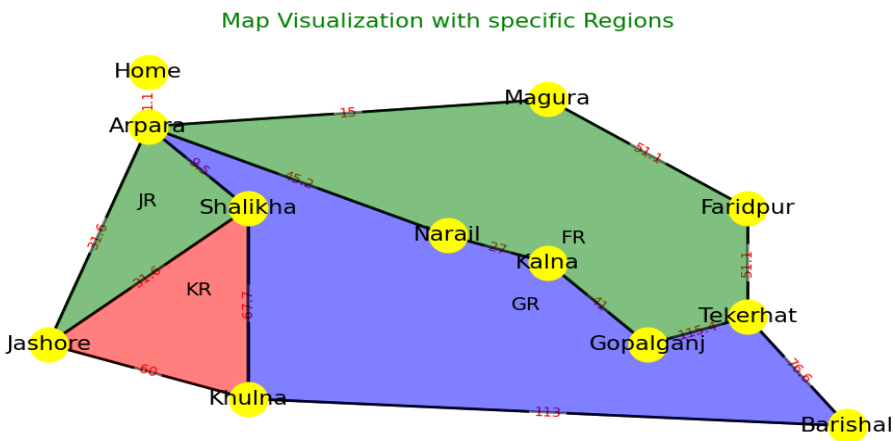

Figure: Map, After coloring

Solutions are complete and consistent assignments, e.g., JR = green, KR = red, GR = blue, and FR = green.

Code implementation in python:

backtracking_search.py

```
# Md. Asad Mondall_20CSE006

import networkx as nx

import matplotlib.pyplot as plt

from matplotlib.patches import Polygon

from collections import Counter

nodes = ["Barishal", "Tekerhat", "Faridpur", "Magura", "Arpara", "Khulna", "Shalikha", "Jashore", "Gopalganj", "Kalna", "Narail", "Home"]

edges = [("Barishal", "Tekerhat", 76.6), ("Tekerhat", "Faridpur", 51.1), ("Faridpur", "Magura", 51.1), ("Magura", "Arpara", 15), ("Arpara", "Home", 1.1),

         ("Khulna", "Shalikha", 67.7), ("Shalikha", "Jashore", 31.6), ("Jashore", "Arpara", 31.6), ("Shalikha", "Arpara", 9.5),("Barishal", "Khulna", 113),("Khulna", "Jashore", 60),

        ("Tekerhat", "Gopalganj", 115.4), ("Gopalganj", "Kalna", 41), ("Kalna", "Narail", 27), ("Narail", "Arpara", 45.2)]

G = nx.Graph()

G.add_nodes_from(nodes)


for edge in edges:

    G.add_edge(edge[0], edge[1], weight=edge[2])

pos = {

    "Barishal": (4, 0),

    "Tekerhat": (3, 4),

    "Faridpur": (3, 8),

    "Magura": (1, 12),

    "Arpara": (-3, 11),

    "Khulna": (-2, 1),

    "Shalikha": (-2, 8),

    "Jashore": (-4, 3),

    "Gopalganj": (2, 3),

    "Kalna": (1, 6),
```

```
    "Narail": (0, 7),
    "Home":(-3, 13)
}
regions = {
    "JR": ["Arpara","Shalikha","Jashore", "Arpara"],
    "KR": ["Shalikha", "Jashore", "Khulna", "Shalikha"],
    "GR": ["Barishal", "Tekerhat", "Gopalganj", "Kalna", "Narail", "Arpara", "Shalikha", "Khulna", "Barishal"],
    "FR": ["Tekerhat", "Gopalganj", "Kalna", "Narail", "Arpara", "Magura", "Faridpur", "Tekerhat"]
}
def is_safe(node, color, graph, color_map):
    for neighbor in graph.neighbors(node):
        if color_map.get(neighbor) == color:
            return False
    return True
def backtrack_coloring(graph, colors, nodes, color_map):
    for node in nodes:
        if node not in color_map:
            for color in colors:
                if is_safe(node, color, graph, color_map):
                    color_map[node] = color
                    break
color_map = {}
for region, nodes in regions.items():
    if region == "KR":
        backtrack_coloring(G, ["red", "blue", "green"], nodes, color_map)
    elif region == "GR":
        backtrack_coloring(G, ["blue", "green", "red"], nodes, color_map)
    else:
        backtrack_coloring(G, ["green", "red", "blue"], nodes, color_map)
plt.figure(figsize=(12, 6), facecolor='white')
nx.draw_networkx_nodes(G, pos, node_size=700, node_color='yellow')
```

```
nx.draw_networkx_edges(G, pos, width=2)
nx.draw_networkx_labels(G, pos, font_size=16, font_color='black', font_family="sans-serif")
edge_labels = {(edge[0], edge[1]): str(edge[2]) for edge in edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')
for region, nodes in regions.items():
    region_nodes = [pos[node] for node in nodes]
    region_colors = [color_map[node] for node in nodes]
    most_common_color = Counter(region_colors).most_common(1)[0][0]
    polygon = Polygon(region_nodes, edgecolor='black', facecolor=most_common_color, closed=True, linewidth=2, alpha=0.5)
    plt.gca().add_patch(polygon)
    region_center = (sum(x for x, y in region_nodes) / len(region_nodes), sum(y for x, y in region_nodes) / len(region_nodes))
    plt.text(region_center[0], region_center[1], region, ha='center', va='center', fontsize=14, color='black')
plt.title("Map Visualization with specific Regions", fontsize=16, color='green')


plt.axis("off")
plt.show()
```

2. Forward Checking

Forward checking is a technique used in constraint satisfaction problems to reduce the search space by pruning the domain of variables.

How Does a Forward Checking Work?

- Keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.
- It propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures

For the same map colorizing problem, it behaves as follows,

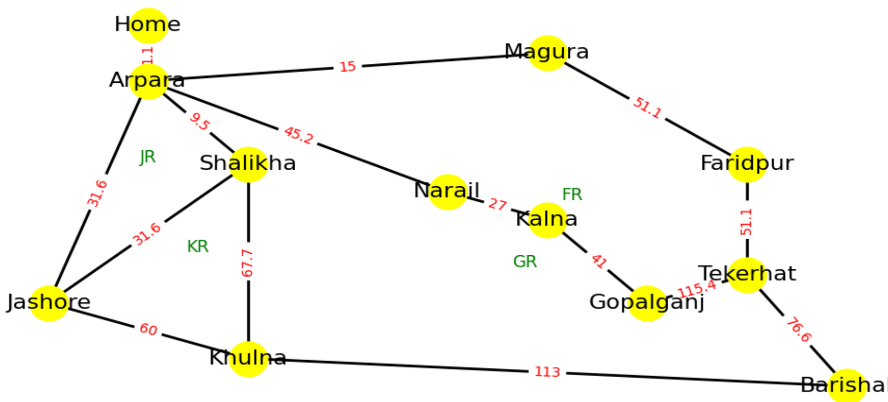

Figure: Map, before colorizing

Variables: JR, KR, GR, FR

JR = Jashore Region, KR = Khulna Region, FR = Faridpur Region

Domains: {red, green, blue}

Constraints: Adjacent regions must have different colors e.g., JR ≠ KR or (JR, KR) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}.

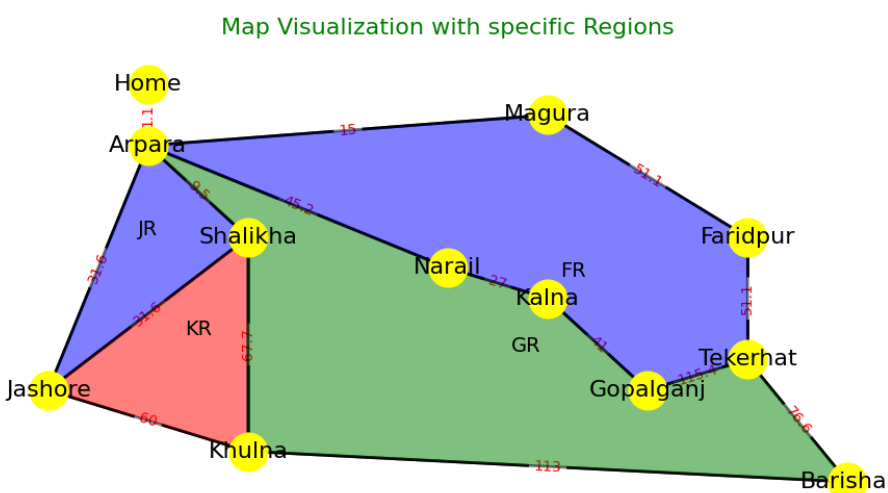

Figure: Map, After coloring

Solutions are complete and consistent assignments, e.g., JR = green, KR = red, GR = blue, and FR = green.

Code implementation in python:

forward_checking.py

```
# Md. Asad Mondall_20CSE006

import networkx as nx

import matplotlib.pyplot as plt

from matplotlib.patches import Polygon


nodes = ["Barishal", "Tekerhat", "Faridpur", "Magura", "Arpara", "Khulna", "Shalikha", "Jashore", "Gopalganj", "Kalna", "Narail", "Home"]

edges = [("Barishal", "Tekerhat", 76.6), ("Tekerhat", "Faridpur", 51.1), ("Faridpur", "Magura", 51.1), ("Magura", "Arpara", 15), ("Arpara", "Home", 1.1),

    ("Khulna", "Shalikha", 67.7), ("Shalikha", "Jashore", 31.6), ("Jashore", "Arpara", 31.6), ("Shalikha", "Arpara", 9.5),("Barishal", "Khulna", 113),("Khulna", "Jashore", 60),

    ("Tekerhat", "Gopalganj", 115.4), ("Gopalganj", "Kalna", 41), ("Kalna", "Narail", 27), ("Narail", "Arpara", 45.2)]


G = nx.Graph()

G.add_nodes_from(nodes)


for edge in edges:

  G.add_edge(edge[0], edge[1], weight=edge[2])


pos = {

  "Barishal": (4, 0),

  "Tekerhat": (3, 4),

  "Faridpur": (3, 8),

  "Magura": (1, 12),

  "Arpara": (-3, 11),

  "Khulna": (-2, 1),

  "Shalikha": (-2, 8),

  "Jashore": (-4, 3),
```

```
    "Gopalganj": (2, 3),
    "Kalna": (1, 5),
    "Narail": (0, 7),
    "Home":(-3, 13)

}

regions = {
    "JR": ["Arpara","Shalikha","Jashore", "Arpara"],
    "KR": ["Shalikha", "Jashore", "Khulna", "Shalikha"],
    "GR": ["Barishal", "Tekerhat", "Gopalganj", "Kalna", "Narail", "Arpara", "Shalikha", "Khulna", "Barishal"],
    "FR": ["Tekerhat", "Gopalganj", "Kalna", "Narail", "Arpara", "Magura", "Faridpur", "Tekerhat"]
}

def forward_checking(graph, colors, nodes, color_map):
    for node in nodes:
        available_colors = set(colors)
        for neighbor in graph.neighbors(node):
            if neighbor in color_map:
                available_colors.discard(color_map[neighbor])
        if available_colors:
            color_map[node] = available_colors.pop()

color_map = {}
for region, nodes in regions.items():
    if region == "JR":
        forward_checking(G, ["blue", "red", "green"], nodes, color_map)
    elif region == "GR":
        forward_checking(G, ["blue", "green", "red"], nodes, color_map)
    else:
        forward_checking(G, ["green", "red", "blue"], nodes, color_map)
```

```
plt.figure(figsize=(12, 6), facecolor='lightgrey')
nx.draw_networkx_nodes(G, pos, node_size=700, node_color='white')
nx.draw_networkx_edges(G, pos, width=2)
nx.draw_networkx_labels(G, pos, font_size=10, font_color='black', font_family="sans-serif")

edge_labels = {(edge[0], edge[1]): str(edge[2]) for edge in edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')

for region, nodes in regions.items():
    region_nodes = [pos[node] for node in nodes]
    region_colors = [color_map[node] for node in nodes]
    polygon = Polygon(region_nodes, edgecolor='black', facecolor=region_colors[0], closed=True, linewidth=2,
alpha=0.5)
    plt.gca().add_patch(polygon)
    region_center = (sum(x for x, y in region_nodes) / len(region_nodes), sum(y for x, y in region_nodes) /
len(region_nodes))
    plt.text(region_center[0], region_center[1], region, ha='center', va='center', fontsize=12, color='black')


plt.title("Map Visualization with specific Regions", fontsize=16, color='green')
plt.axis("off")
plt.show()
```

3. Arc Consistency

Arc consistency is a property that ensures that every value in the domain of a variable is consistent with the constraints imposed by other variables in a constraint satisfaction problem (CSP). Arc consistency in graph coloring ensures that for every pair of neighboring nodes, there exists at least one color in the domain of each node that satisfies the constraint of different colors for adjacent nodes. By enforcing arc consistency, the search space for finding a valid coloring of the graph is reduced, which can help speed up the coloring process and improve efficiency.

How Does a Forward Checking Work?

1. **Initialization**: Each node in the graph has an associated domain of colors, typically represented as a list of possible colors that can be assigned to that node. Initially, all nodes have their domains set to include all available colors.
2. **Enforcing Arc Consistency (AC-3):**
    - Start with a queue containing all arcs (pairs of neighboring nodes) in the graph.
    - Repeat the following steps until the queue is empty:
        - Dequeue an arc (Xi, Xj) from the queue.
        - Check if the domain of Xi needs to be revised based on the constraint that Xi and Xj must have different colors.
        - If Xi's domain is revised (i.e., a color is removed), and it becomes empty, return failure, indicating that the graph cannot be colored consistently.
        - If Xi's domain is revised, and it still has colors remaining, enqueue all arcs (Xk, Xi), where Xk is a neighbor of Xi and Xk ≠ Xj.
3. **Termination:**
    - If the AC-3 algorithm completes without any domain becoming empty, the graph can be colored consistently.

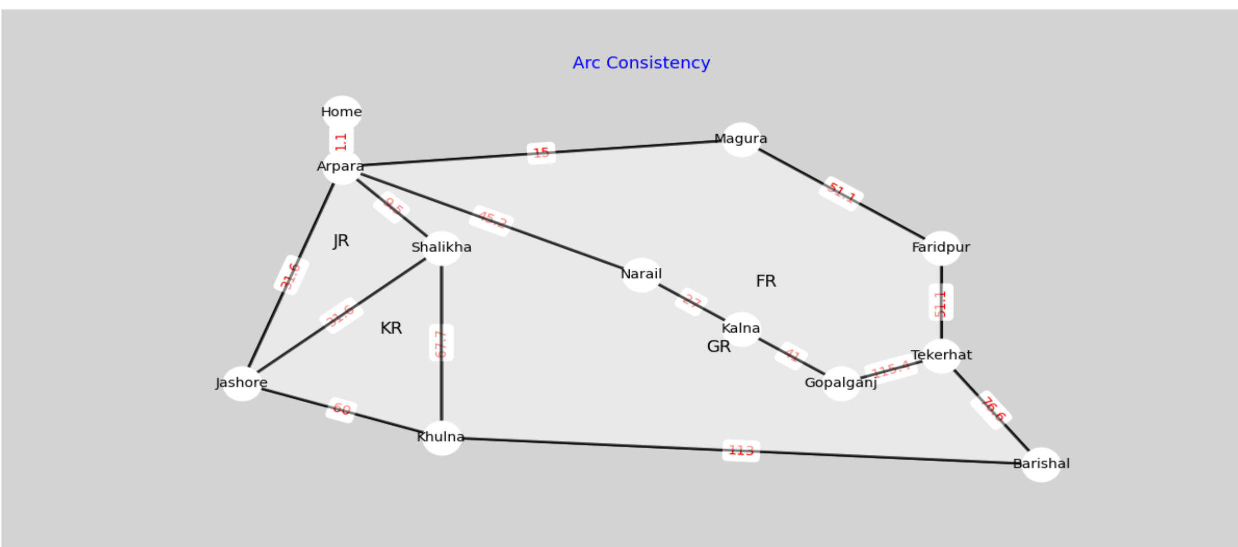

Figure: Map, after applying Arc Consistency

Code implementation in python:

arc_consistency.py

```
# Md. Asad Mondall_20CSE006

import networkx as nx

import matplotlib.pyplot as plt

from matplotlib.patches import Polygon

from collections import Counter

from queue import Queue


# Define the nodes and edges

nodes = ["Barishal", "Tekerhat", "Faridpur", "Magura", "Arpara", "Khulna", "Shalikha", "Jashore", "Gopalganj", "Kalna", "Narail", "Home"]

edges = [("Barishal", "Tekerhat", 76.6), ("Tekerhat", "Faridpur", 51.1), ("Faridpur", "Magura", 51.1), ("Magura", "Arpara", 15), ("Arpara", "Home", 1.1),

         ("Khulna", "Shalikha", 67.7), ("Shalikha", "Jashore", 31.6), ("Jashore", "Arpara", 31.6), ("Shalikha", "Arpara", 9.5),("Barishal", "Khulna", 113),("Khulna", "Jashore", 60),

        ("Tekerhat", "Gopalganj", 115.4), ("Gopalganj", "Kalna", 41), ("Kalna", "Narail", 27), ("Narail", "Arpara", 45.2)]


# Create a graph

G = nx.Graph()


# Add nodes to the graph

G.add_nodes_from(nodes)


# Add edges to the graph

for edge in edges:

   G.add_edge(edge[0], edge[1], weight=edge[2])


# Define positions of nodes with padding

pos = {
```

```
    "Barishal": (4, 0),
    "Tekerhat": (3, 4),
    "Faridpur": (3, 8),
    "Magura": (1, 12),
    "Arpara": (-3, 11),
    "Khulna": (-2, 1),
    "Shalikha": (-2, 8),
    "Jashore": (-4, 3),
    "Gopalganj": (2, 3),
    "Kalna": (1, 5),
    "Narail": (0, 7),
    "Home":(-3, 13)

}

# Define regions
regions = {
    "JR": ["Arpara","Shalikha","Jashore", "Arpara"],
    "KR": ["Shalikha", "Jashore", "Khulna", "Shalikha"],
    "GR": ["Barishal", "Tekerhat", "Gopalganj", "Kalna", "Narail", "Arpara", "Shalikha", "Khulna", "Barishal"],
    "FR": ["Tekerhat", "Gopalganj", "Kalna", "Narail", "Arpara", "Magura", "Faridpur", "Tekerhat"]
}

# Backtracking algorithm to check if assigning a color to a node is safe
def is_safe(node, color, graph, color_map):
    for neighbor in graph.neighbors(node):
        if color_map.get(neighbor) == color:
            return False
    return True

# AC-3 algorithm to enforce arc consistency
```

```
def ac3(graph, domains):
    queue = Queue()
    for edge in graph.edges:
        queue.put(edge)
    while not queue.empty():
        (Xi, Xj) = queue.get()
        if revise(graph, domains, Xi, Xj):
            if len(domains[Xi]) == 0:
                return False
            for Xk in graph.neighbors(Xi):
                if Xk != Xj:
                    queue.put((Xk, Xi))
    return True


def revise(graph, domains, Xi, Xj):
    revised = False
    for color in domains[Xi]:
        if not any(is_safe(Xi, color, graph, {Xi: color, Xj: c}) for c in domains[Xj]):
            domains[Xi].remove(color)
            revised = True
    return revised


# Forward checking algorithm to color the regions
def forward_checking(graph, colors, nodes, domain, color_map):
    if len(nodes) == 0:
        return True
    node = nodes[0]
    for color in domain[node]:
        if is_safe(node, color, graph, color_map):
            color_map[node] = color
            domain_copy = domain.copy()
```

```
        for neighbor in graph.neighbors(node):
            if color in domain_copy[neighbor]:
                domain_copy[neighbor].remove(color)
        if ac3(graph, domain_copy):
            if forward_checking(graph, colors, nodes[1:], domain_copy, color_map):
                return True
        color_map.pop(node)
    return False

# Color the regions using forward checking
color_map = {}
domain = {node: ["red", "blue", "green"] for node in nodes}
forward_checking(G, ["red", "blue", "green"], nodes, domain, color_map)

# Draw the graph with curved layout
plt.figure(figsize=(12, 6), facecolor='lightgrey')  # Set background color and size

# Draw nodes
nx.draw_networkx_nodes(G, pos, node_size=700, node_color='white')

# Draw edges
nx.draw_networkx_edges(G, pos, width=2)

# Draw labels with specified color
nx.draw_networkx_labels(G, pos, font_size=10, font_color='black', font_family="sans-serif")

# Add edge labels with specified color
edge_labels = {(edge[0], edge[1]): str(edge[2]) for edge in edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')

# Draw regions and label in the center
```

```
for region, nodes in regions.items():
    region_nodes = [pos[node] for node in nodes]
    most_common_color = Counter([color_map.get(node, 'white') for node in nodes]).most_common(1)[0][0]
    polygon = Polygon(region_nodes, edgecolor='black', facecolor=most_common_color, closed=True, linewidth=2, alpha=0.5)
    plt.gca().add_patch(polygon)
    region_center = (sum(x for x, y in region_nodes) / len(region_nodes), sum(y for x, y in region_nodes) / len(region_nodes))
    plt.text(region_center[0], region_center[1], region, ha='center', va='center', fontsize=12, color='black')

# Set the title with bright font color
plt.title("Arc Consistency", color='blue')

# Show the graph
plt.axis("off")
plt.show()
```